White Paper

# Advanced Encryption Standard (AES) Instructions Set

Intel Mobility Group
Israel Development Center, Israel

Shay Gueron

Intel's AES-NI is a new set of Single Instruction Multiple Data (SIMD) instructions that are going to be introduced in the next generation of Intel® processor, as of 2009. These instructions enable fast and secure data encryption and decryption, using the Advanced Encryption Standard (AES), defined by FIPS Publication number 197.

The architecture introduces six instructions that offer full hardware support for AES. Four of them support high performance data encryption and decryption, and the other two instructions support the AES key expansion procedure.

This white paper provides an overview of the AES algorithm and guidelines for utilizing the AES instructions to achieve secure high performance AES processing.

*April 2008*

Intel Corporation

# Contents



## Figures

# Introduction

The Advanced Encryption Standard (AES) is the United States Government standard for symmetric encryption, defined by FIPS Publication #197 (2001). It is used in a large variety of applications, and high throughput and security are required.

Intel offers a new set of Single Instruction Multiple Data (SIMD) instructions that will be introduced in the next generation of the Intel® processor family. These instructions enable fast and secure encryption and decryption using AES.

The new architecture introduces six Intel® SSE instructions. Four instructions, namely AESENC, AESENCLAST, AESDEC, and AESDELAST facilitate high performance AES encryption and decryption. The other two, namely AESIMC and AESKEYGENASSIST, support the AES key expansion procedure. Together, these instructions provide a full hardware for support AES, offering security, high performance, and a great deal of flexibility.

In addition, all six instructions are promoted to a GSSE-128 version, as a non-destructive destination instructions, namely VAESENC, VAESENCLAST, VAESDEC, VAESDELAST, VAESIMC, and VAESKEYGENASSIST.

This white paper provides an overview of the AES algorithm and guidelines for utilizing the AES instructions to achieve high performance and secure AES processing. Some special usage models of this architecture are also described.

# Preliminaries: AES and Intel® Architecture

## AES Definition and Brief Description

The Advanced Encryption Standard (AES) is the United States Government standard for symmetric encryption, defined by FIPS Publication #197 (FIPS197 hereafter).

AES is a block cipher that encrypts a 128-bit block (plaintext) to a 128-bit block (ciphertext), or decrypts a 128-bit block (ciphertext) to a 128-bit block (plaintext).

AES uses a key (cipher key) whose length can be 128, 192, or 256 bits. Hereafter encryption/decryption with a cipher key of 128, 192, or 256 bits is denoted AES-128, AES192, AES-256, respectively.

AES-128, AES-192, AES-256 process the data block in, respectively, 10, 12, or 14 iterations of a pre-defined sequence of transformations, which are also called "rounds" (AES rounds) for short. The rounds are identical except for the last one, which slightly differs from the others (by skipping one of the transformations).

The rounds operate on two 128-bit inputs: "State" and "Round key". Each round from 1 to 10/12/14 uses a different Round key. The 10/12/14 round keys are derived from the cipher key by the "Key Expansion" Algorithm. This algorithm is independent of the processed data, and can be carried out independently of the encryption/decryption phase.

The data block is processed serially as follows: initially, the input data block is XOR-ed with the first 128 bits of the cipher key to generate the "State". This step is also referred to as "Round 0" which is using round key #0 (round key #0 is the first 128 bits

of the cipher key). Subsequently, the State is serially passed through 10/12/14 rounds where the result of the last round is the encrypted (decrypted) block.

# Little Endian Intel® Architecture and Big Endian FIPS197 Specification

FIPS197 describes the AES algorithm, and provides test vectors while following a Big Endian format convention. On the other hand, Intel's architecture follows a Little Endian format convention. The proper translation between these notations is required. In particular, translating the FIPS197 test vectors to Intel's Little Endian notation requires byte-reflection. To illustrate, the encoding of a 128-bit vector in FIPS197 notation (read from left to right) is [Byte0, byte1, …, Byte14, Byte15]. Inside the bytes, the encoding is Little Endian, where the leftmost bit is the most significant bit. Thus, the bit-wise encoding is as follows: [7-0, 15-8, 23-16, 31-24, …127-120].

To translate the vector to an IA compatible Little Endian format, it needs to be byte-reflected to [Byte15, byte14, …, Byte1, Byte0], so that the resulting bit-wise encoding becomes [127-120, … 31-24, 23-16, 15-8, 7-0].

Hereafter, 128-bit vectors are written in a hexadecimal notation, where each byte is denoted as a two-digit (hexadecimal) number.

## Example

Consider the vector d4bf5d30e0b452aeb84111f11e2798e5 written in the FIPS197 notation. This notation represents 16 2-digit numbers (hexadecimal notation) as follows: "d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5". Here, d4 as the least significant byte.

When this vector is written in an IA-compatible format, it reads e590271ef11141b8ae52b4e0305dbfd4. This corresponds to 16 2-digit numbers (hexadecimal notation) "e5 90 27 1e f1 11 41 b8 ae 52 b4 e0 30 5d bf d4". The corresponding 128-bit encoding is:

11100101100100000010011100011110111100010001000101000001101110001010111001010010101101001110000000011000001011101101111111101010100

(With the most significant bit equals 1 and the least significant bit equals 0).

# AES Data Structure in Terms of IA

Intel's AES instructions operate on one or on two 128-bit inputs with the typical instruction format being "instruction xmm1 xmm2/m128". Here, xmm1 xmm2 are aliases to any two xmm registers, and the result is written into xmm1. The /m128 indicates a register-memory instruction.

When referring to the contents of an xmm register, one may refer to the bits (127-0), the bytes (15-0), or the 32-bit double words (doublewords) (3-0). The bytes are also referred to by the characters P-A and the doublewords are referred to by X3-X0. Figure 1 illustrates the corresponding bit/byte/doublewords positions.

**Figure 1. State Bit, Byte, and Doubleword Positions in an xmm Register**

| 127-120 | 119-112 | 111-104 | 103-96 | 95-88 | 87-08 | 79-72 | 71-64 | 63-56 | 55-48 | 47-40 | 39-32 | 31-24 | 23-16 | 15-8 | 7-0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 3 (127-96) | | | | 2 (95-64) | | | | 1 (63-32) | | | | 0 (31-0) | | | |
| **X3** | | | | **X2** | | | | **X1** | | | | **X0** | | | |
| **P** | **O** | **N** | **M** | **L** | **K** | **J** | **I** | **H** | **G** | **F** | **E** | **D** | **C** | **B** | **A** |

| **4x4 Matrix Notation of an xmm Register** | | | |
|---|---|---|---|
| A | E | I | M |
| B | F | J | N |
| C | G | K | O |
| D | H | L | P |

## Example

FIPS197 vector d4bf5d30e0b452aeb84111f11e2798e5 (in Big Endian format) has the least significant byte d4 and the significant byte e5. When it is translated to the Little Endian IA format, using the P-A notation, it becomes:

| Byte # | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A |
| | e5 | 98 | 27 | 1e | f1 | 11 | 41 | b8 | ae | 52 | b4 | e0 | 30 | 5d | bf | d4 |

The corresponding 4x4 matrix format is:

| **4x4 Matrix Format of an xmm Registerh** | | | | **The FIPS197 Vector Arranged in the 4x4 Format** | | | |
|---|---|---|---|---|---|---|---|
| A | E | I | M | d4 | E0 | b8 | 1e |
| B | F | J | N | Bf | B4 | 41 | 27 |
| C | G | K | O | 5d | 52 | 11 | 98 |
| D | H | L | P | 30 | ae | f1 | e5 |

# The AES Algorithm

This chapter describes the functions and the transformations that are underline the definition of AES. Note that some of these transformations are expressed here in a Little Endian format, and not as in the FIPS 197 document.

## Cipher Key

AES is a symmetric key encryption algorithm. It uses a cipher key whose length is 128 bits, 192 bits or 256 bits. The algorithm with cipher key length 128, 192, 256 bits is denoted AES-128, AES-192, AES-256, respectively.

## State

The process of encrypting (decryption) of plaintext (ciphertext) to ciphertext (plaintext) generates intermediate 128-bit results. These intermediate results are referred to as the State.

## Data Blocks

AES operates on an input data block of 128 bits and its output is also a data block of 128 bits. During the process, the data block is call a State.

## Round Keys

AES-128, AES192, AES-256 algorithm expends the a cipher key 10, 12, 14 round keys from, respectively. Each Round key is 128-bit in length. The algorithm for deriving the round keys from the cipher key is the called Key Expansion.

## AddRoundKey

AddRoundKey is a (128-bit, 128-bit) → 128-bit transformation, which is defined, is a bit-wise xor of its two arguments. In the AES flow, the arguments are the State and the round key.

## S-Box and InvS-Sbox

S-Box (Substitution Box) is an 8-bit → 8-bit transformation defined as the affine function x → A $x^{-1}$ + b where A is an 8x8 binary matrix and b is an 8-bit vector as follows:

$$\begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{pmatrix} \rightarrow \begin{pmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{pmatrix} \begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{pmatrix}^{-1} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Here $(\ )^{-1}$ denotes inversion is in the Galois Field (Finite Field) GF($2^8$) defined by the polynomial $x^8+x^4+x^3+x+1$ (0x11b for short). Hereafter, this field is referred to as AES-GF256-Field.

***Note:*** Matrix A and the vector b are accommodated to the Little Endian notation (therefore not identical to the corresponding values in FIPS197).

InvS-Box is the inverse transformation of S-Box, defined as $y \rightarrow (A^{-1}\ y + A^{-1}\ b)^{-1}$, which is computed to be:

$$\begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{pmatrix} \rightarrow \left( \begin{pmatrix} 10101001 \\ 00101001 \\ 10010100 \\ 01001010 \\ 00100101 \\ 10010010 \\ 01001001 \\ 10100100 \end{pmatrix} \begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \right)^{-1}$$

## S-Box and InvS-Box Lookup Tables

The S-Box and InvS-Box transformations can be represented via a lookup table as follows. The input to the lookup tables is a byte B [7-0] where x and y denote it low and high nibbles x [3-0] = B [7-4], y [3-0] = B [3-0].The output byte is encoded in the table as a two digits number in hexadecimal notation. For example, S-Box lookup for the input 85 (x=8; y=5 in hexadecimal notation) yields 97 in hexadecimal notation. InvS-Box lookup for the input 97 yields 85.

**Figure 2. S-Box and InvS-Box Lookup Tables**

S-Box lookup table

```
              <------------------ y ---------------------->
                0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
           ^ 0 63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
           | 1 ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
           | 2 b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
           | 3 04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
           | 4 09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
           | 5 53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
             6 d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
           x 7 51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
             8 cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
           | 9 60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
           | a e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
           | b e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
           | c ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
           | d 70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
           | e e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
           V f 8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
```

InvS-Box lookup table

```
              <-------------------- y ---------------------->
 1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
          ^ 0 52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
          | 1 7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
          | 2 54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
          | 3 08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
          | 4 72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
          | 5 6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
            6 90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
          x 7 d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
            8 3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
          | 9 96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
          | a 47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
          | b fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
          | c 1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
          | d 60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
          | e a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
          V f 17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```

## SubBytes Transformation

SubBytes are the 16-bytes → 16-bytes (byte-wise) transformation defined by applying the S-Box transformation to each one of the 16 bytes of the input, namely:

[P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [S-Box (P), S-Box (O), S-Box (N), S-Box (M), S-Box (L), S-Box (K), S-Box (J), S-Box (I), S-Box (H), S-Box (G), S-Box (F), S-Box (E), S-Box (D), S-Box (C), S-Box (B), S-Box (A)].

### SubBytes Example

SubBytes (73744765635354655d5b56727b746f5d) =
8f92a04dfbed204d4c39b1402192a84c

## InvSubBytes Transformation

InvSubBytes is a 16-byte → 16-byte (byte-wise) transformation defined by applying the InvS-Box function to each byte of the input, namely:

[P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [InvS-Box (P), InvS-Box (O), InvS-Box (N), InvS-Box (M), InvS-Box (L), InvS-Box (K), InvS-Box (J), InvS-Box (I), InvS-Box (H), InvS-Box (G), InvS-Box (F), InvS-Box (E), InvS-Box (D), InvS-Box (C), InvS-Box (B), InvS-Box (A)].

### InvSubBytes Example

InvSubBytes (5d7456657b536f65735b47726374545d) =
8dcab9dc035006bc8f57161e00cafd8d

## ShiftRows Transformation

ShiftRows is the following byte-wise permutation: (15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0) → (11, 6, 1, 12, 7, 2, 13, 8, 3, 14, 9, 4, 15, 10, 5, 0). In the P-A notation is reads [P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [L,G,B,M,H,C,N,I,D,O,J,E,P,K,F,A]. Its

name comes from viewing the transformation as an operation on the 4x4 matrix representation of the state. The first row is unchanged. The second row is left rotated by one byte position. The third row is left rotated by two byte positions. The fourth row is left rotated by three byte positions.

### ShiftRows Example

ShiftRows (7b5b546573745666563746f725d53475d) =
737447656353546555d5b56727b746f5d

## InvShiftRows Transformation

InvShiftRows is the inverse of ShiftRows. It is the following byte-wise permutation: (15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0) → (3, 6, 9, 12, 15, 2, 5, 8, 11, 14, 1, 4, 7, 10, 13, 0). In the P-A notation is reads [P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [D,G,J,M,P,C,F,I,L,O,B,E,H,K,N,A]

### InvShiftRows Example

InvShiftRows (7b5b546573745666563746f725d53475d) =
5d7456657b536f65735b47726374545d

## MixColumns Transformation

MixColumns is a 128-bit → 128-bit transformation operating on the columns of the 4x4 matrix representation of the input. The transformation treats each column as a third degree polynomial with coefficients in AES-GF256-Field. Each column of the 4x4 matrix representation of the state is multiplied by polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ and reduced modulo $x^4 + 1$. Here, {--} denotes an element in AES-GF256-Field. The equations that define MixColumns are detailed in Figure 3. The transformation is [P – A] → [P' – A']; the symbol • denotes multiplication in AES-GF256-Field (i.e., • is a carry-less multiplications followed by reduction mod 0x11b); the symbol + denotes XOR.

**Figure 3. MixColumns Transformation Equations**

```
A' = ({02} • A) + ({03} • B) + C + D
B' = A + ({02} • B) + ({03} • C) + D
C' = A + B + ({02} • C) + ({03} • D)
D' = ({03} • A) + B + C + ({02} • D)
E' = ({02} • E) + ({03} • F) + G + H
F' = E + ({02} • F) + ({03} • G) + H
G' = E + F + ({02} • G) + ({03} • H)
H' = ({03} • E) + F + G + ({02} • H)
I' = ({02} • I) + ({03} • J) + K + L
J' = I + ({02} • J) + ({03} • K) + L
K' = I + J + ({02} • K) + ({03} • L)
L' = ({03} • I) + J + K + ({02} • L)
M' = ({02} • M) + ({03} • N) + O + P
N' = M + ({02} • N) + ({03} • O) + P
O' = M + N + ({02} • O) + ({03} • P)
P' = ({03} • M) + N + O + ({02} • P)
```

### MixColumns Example

MixColums (627a6f6644b109c82b18330a81c3b3e5) =
7b5b54657374566563746f725d53475d

## InvMixColumns Transformation

InvMixColumns is a 128-bit → 128-bit transformation operating on the columns of the 4x4 matrix representation of the input. The transformation treats each column as a third degree polynomial with coefficients in AES-GF256-Field. Each column of the 4x4 matrix representation of the state is multiplied by polynomial $a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$ and reduced modulo $x^4 + 1$. The equations that define InvMixColumns are detailed in Figure 4. The transformation is [P – A] → [P' – A']; the symbol • denotes multiplication in AES-GF256-Field (i.e., • is a carry-less multiplications followed by reduction mod 0x11b); the symbol + denotes XOR.

**Figure 4. InvMixColumns Transformation Equations**

```
A' = ({0e} • A) + ({0b} • B) + ({0d} • C) + ({09} • D)
B' = ({09} • A) + ({0e} • B) + ({0b} • C) + ({0d} • D)
C' = ({0d} • A) + ({09} • B) + ({0e} • C) + ({0b} • D)
D' = ({0b} • A) + ({0d} • B) + ({09} • C) + ({0e} • D)
E' = ({0e} • E) + ({0b} • F) + ({0d} • G) + ({09} • H)
F' = ({09} • E) + ({0e} • F) + ({0b} • G) + ({0d} • H)
G' = ({0d} • E) + ({09} • F) + ({0e} • G) + ({0b} • H)
H' = ({0b} • E) + ({0d} • F) + ({09} • G) + ({0e} • H)
I' = ({0e} • I) + ({0b} • J) + ({0d} • K) + ({09} • L)
J' = ({09} • I) + ({0e} • J) + ({0b} • K) + ({0d} • L)
K' = ({0d} • I) + ({09} • J) + ({0e} • K) + ({0b} • L)
L' = ({0b} • I) + ({0d} • J) + ({09} • K) + ({0e} • L)
M' = ({0e} • M) + ({0b} • N) + ({0d} • O) + ({09} • P)
N' = ({09} • M) + ({0e} • N) + ({0b} • O) + ({0d} • P)
O' = ({0d} • M) + ({09} • N) + ({0e} • O) + ({0b} • P)
P' = ({0b} • M) + ({0d} • N) + ({09} • O) + ({0e} • P)
```

### InvMixColumns example

InvMixColumns (8dcab9dc035006bc8f57161e00cafd8d) =
5be3eb11928b5eaeeec9cc3bc55f5777

## SubWord Transformation

SubWord is the doubleword → doubleword transformation defined by applying the S-Box transformation to each one of the 4 bytes of the input, namely:

SubWord (X) = [S-Box(X[31-24]), S-Box(X[23-16]), S-Box(X[15-8]), S-Box(X[7-0])]

### SubWord Example

SubWord (73744765) = 8f92a04d

# RotWord Transformation

RotWord is the doubleword → doubleword transformation defined by:

RotWord (X [31-0]) = [X[7-0], X [31-24], X [23-16], X [15-8]]

(in C language notation, RotWord(X) = (X >> 8) | (X << 24))

## RotWord Example

RotWord (3c4fcf09) = 093c4fcf

# Round Constant (RCON)

AES Key Expansion procedure uses ten constants called Round Constants (denoted RCON hereafter). The ten RCON values are RCON [i] = $\{02\}^{i-1}$ for i=1, 2, …, 10, where the operations are in AES-GF256-Field.

Each RCON value is an element of AES-GF256-Field, and is encoded here as a byte. The ten RCON values (written in hexadecimal notation) are:

RCON [1] = 01, RCON [2] = 02, RCON [3] = 04, RCON [4] = 08, RCON [5] = 10,

RCON [6] = 20, RCON [7] = 40, RCON [8] = 80, RCON [9] = 1B, RCON [10] = 36.

In the following RCON values are also perceived as doublewords where the 24 most significant bit equal 0. For example, RCON [7] = 00000040 (written in hexadecimal notation).

# Key Expansion

AES uses a key (cipher key) whose length is 128, 192 or 256 bits. The cipher key is expanded to into 10, 12, or 14 round keys, respectively, using the "Key Expansion" Algorithm, where each round key is of 128 bits. This Key Expansion algorithm depends only on the cipher key, and is independent of the processed data. It can therefore be executed independently of the encryption/decryption phase. At the heart of the algorithm for generating the round keys are the combination of transformations SubWord(RotWord(tmp)) and SubWord(tmp) and the use of the RCON value. The AES Key Expansion algorithm is described by the pseudo code in Figure 5 (the pseudo code is written in terms of doublewords). Note that this description of the Key Expansion algorithm is Big Endian oriented.

**Figure 5. AES key Expansion Pseudo Code (Described in FIP197)**

```
Parameters
Nb = 4
Nk = number of doublewords in the cipher key (4, 6, 8 for AES-128, AES-192, AES-256,
resp.)
Nr = number of rounds in the cipher (Nr=10, 12, 14 AES-128, AES-192, AES-256, resp.)

The KeyExpansion routine

KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
word tmp
i = 0
while (i < Nk)
```

```
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
    i = i+1
end while
i = Nk
while (i < Nb * (Nr+1)]
    tmp = w[i-1]
    if (i mod Nk = 0)
        tmp = SubWord(RotWord(tmp)) xor RCON[i/Nk]
    else
    if (Nk > 6 and i mod Nk = 4)
        tmp = SubWord(tmp)
    end if
    w[i] = w[i-Nk] xor tmp
    i = i + 1
end while
```

# AES Encryption and Decryption Flows

## The Order of Transformations

SubBytes transformation operates separately on each byte of the State, where ShiftRows transformation operates on the columns of the State. Therefore, ShiftRows and SubBytes commute with each other. Similarly, InvShiftRows and InvSubBytes commute.

In the following, ShiftRows is applied before SubBytes and InvShiftRows is applied before InvSubBytes.

## The Equivalent Inverse Cipher for Decryption

There are two equivalent ways to perform AES decryption, one is called the "Inverse Cipher" and the other is called the "Equivalent Inverse Cipher". They differ in the internal order of the sequence of (inverse) transformations, and also in the way that the decryption round keys are defined.

Intel architecture follows the Equivalent Inverse Cipher for decryption.

The advantage of using the Equivalent Inverse Cipher is that the order of transformations in the encryption and the decryption flows is identical (except for the fact that the transformations are inversed in decryption).

To use the "Equivalent Inverse Cipher", the decryption round keys must be prepared accordingly. Specifically, the decryption round keys are the 10/12/14 encryption after these are passed through (each one separately) the InvMixColumns transformation. During decryption, decryption round keys are, of course, consumed in reverse order compared with the encryption round keys.

## Encryption and Decryption flows

AES encryption and decryption flows use the expanded key (recall that key expansion is independent of the processed data). The encryption/decryption procedure is a back-to-back sequence of AES transformations, operating on a 128-bit State (data) and a round key. The actual flows depend on the cipher key length, where

**AES-128** encryption/decryption consists of 40 steps

**AES-192** encryption/decryption consists of 48 steps

**AES-256** encryption/decryption consists of 56 steps

The color code is used in Figure 6 and Figure 7 for describing the respective different flows. In the following pseudo code, it is assumed that the **10**, **12**, or **14** round keys are already derived (expanded) from the cipher key and stored, in the proper order, in an array (Round_Key_Encrypt and Round_Key_Decrypt). More precisely, it is assumed that Round_Key_Encrypt [0] stores the first 128 bits of the cipher key (for "round #0" which is a simple xor operation).

**Figure 6. The AES Encryption Flow**

```
;; Data is a 128-bit block to be encrypted. The round keys are stored in
Round_Key_Encrypt

     Tmp = Add Round Key (Data, Round_Key_Encrypt [0])
     For round =  1-9 or 1-11 or 1-13:
       Tmp = ShiftRows  (Tmp)
       Tmp = SubBytes (Tmp)
       Tmp = MixColumns (Tmp)
       Tmp = AddRoundKey (Tmp, Round_Key_Encrypt [round])
     end loop
     Tmp = ShiftRows (Tmp)
     Tmp = SubBytes (Tmp)
     Tmp = Add RoundKey (Tmp, Round_Key_Encrypt [10 or 12 or 14])
     Result = Tmp
```

**Figure 7. The AES Decryption Flow (Using the Equivalent Inverse Cipher)**

```
;; Data is a 128-bit block to be decrypt. The round keys are stored in
Round_Key_Decrypt

     Tmp = Add Round Key (Data, Round_Key_Decrypt [0])
     For round =  1-9 or 1-11 or 1-13:
       Tmp = InvShiftRows  (Tmp)
       Tmp = InvSubBytes (Tmp)
       Tmp = InvMixColumns (Tmp)
       Tmp = AddRoundKey (Tmp, Round_Key_Decrypt [round])
     end loop
     Tmp = InvShift Rows (Tmp)
     Tmp = InvSubBytes (Tmp)
     Tmp = AddRoundKey (Tmp, Round_Key_Decrypt [10 or 12 or 14])
     Result = Tmp
```

# Software Side Channels

This chapter provides a brief description of software side channel attacks and explains why memory access patterns can be used against software implementations of AES that use table lookups.

## What are Software Side Channel Attacks?

Software side channels are a set of vulnerabilities targeting modern computing environments. They can be potentially used for attacking cryptographic applications that run on a multi-tasking platform.

These attacks use the fact that virtually all current commercial computer platforms run multiple tasks on a single set of hardware resources. Indeed, some recent publications showed that multi-tasking operating systems combined with processor's resource sharing can lead to side channel information leaks where an un-privileged spy process running in parallel to some cryptosystem (crypto) and can infer information on crypto's memory access patterns or on its execution flow.

The focus of this chapter is on software side channel, and how attacks on AES are mitigated by using the AES instructions.

## The CPU Cache and The Basics Of Cache Attacks

Cache is a widely used performance optimizations technique in employed by all modern processors. The cache is a special (and expensive) type of memory that allow for very fast access – much faster than access to the main memory. This memory is used by the CPU for storing the recently read areas of memory. This way, for each memory access, the CPU first checks if the required data is already in the cache. In that case (called cache hit), the memory access is very fast. If the required data is not in the cache (case called cache miss), it is read, more slowly, from the memory, and stored in the cache for future calls. Obviously, storing new data in the cache requires that the CPU evict some other previously loaded data (typically the less recent data) to make room. As a result of utilizing the cache, the average memory access time is significantly reduced. However, the time it take a particular piece of data to be accessed, depends on whether or not this data is already in the cache – and malicious code could potentially exploit this cache behavior if the cryptographic application has sensitive data dependent memory access patterns.

## Table Lookups and the Implied Vulnerability

Currently, the most efficient AES software implementations use large lookup tables (e.g., Gladmans implementation http://fp.gladman.plus.com/ or the OpenSSL code http://www.openssl.org). Typically, AES software uses five lookup tables. The access to these tables depends upon the secret key and the data that is being encrypted or decrypted – opening a side channel.

A potential attack has a spy process executing on the same system as the target crypto process. The spy fills the cache with its own data and then reads it back. By measuring it own read latency, the spy can identify cache lines which were evicted, and this can be used to deduce which parts of the table were accessed by the AES application. Analysis of this information can lead to revealing the secret key (in fact, the first and last rounds have the most opportunity to leak information about the key due to knowledge of which cache lines were accessed). For details on side channel attack on software implementation of AES, see for example (among many references): D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES", Lecture Notes in Computer Science series, Springer-Verlag, 3860: 1-20, (2006) and also D. J. Bernstein, "Cache-timing attacks on AES", http://people.csail.mit.edu/tromer/papers/cache.pdf (2005).

## Software Mitigation for AES Carries a Performance Penalty

There are ways to write an AES software implementation that eliminates significant key and data dependent memory accesses of the AES algorithm. One example is to permute the lookup tables periodically. However, mitigation techniques are not free, and in the case of AES, they carry a significant performance penalty. Some details can be found in the paper: E. Brickell, E., G. Graunke, M. Neve, J. P. Seifert, "Software mitigations to

hedge AES against cache based software side channel vulnerabilities" http://eprint.iacr.org/2006/052.pdf. Another way is to write the AES software without accessing lookup tables (Bit Slicing), but this also involves a significant performance penalty. Details can be found in the paper: M. Matsui and S. Fukuda. "How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors". LNCS, Springer Verlag, 3557: 398–412 (2005).

## The AES Instructions Protect AES Against Side Channels Attacks

The new Intel instructions set offer fast AES encryption/decryption, which is protected against side channels attacks. All of the instructions have fixed and data-independent latency. Furthermore, all the computations are performed completely by hardware. Therefore, using the AES instructions, AES encryption/decryption and key expansion has fixed time and involves no memory access.

As a result, the potential threat of abusing the system and extracting secret information from memory access patterns is eliminated.

# Intel's AES Architecture

The AES instructions set consists of six instructions.

Four instructions, namely AESENC, AESENCLAST, AESDEC, AESDECLAST, are provided for data encryption and decryption. The names are, respectively, short for AES Encrypt Round, AES Encrypt Last Round, AES Decrypt Round AES Decrypt Last Round. These instructions have register-register and register-memory variants.

Two other instructions, namely AESIMC and AESKEYGENASSIST are provided in order to assist with AES key expansion. The names are, respectively, short for AES Inverse Mix Columns, and AES Key Generation Assist.

## The Four AES Rounds Instructions

AESENC, AESENCLAST, AESDEC, AESDECLAST are defined by the pseudo codes detailed in the following figures. Note that these instructions perform a set of grouped transformations that correspond to the AES encryption/decryption flow described in Figure 8 and Figure 9.

**Figure 8. The AESENC and AESENCLAST Instructions**

```
AESENC xmm1, xmm2/m128                AESENCLAST xmm1, xmm2/m128
Tmp := xmm1;                          Tmp := xmm1;
Round Key := xmm2/m128;               Round Key := xmm2/m128;
Tmp := ShiftRows (Tmp);               Tmp := Shift Rows (Tmp);
Tmp := SubBytes (Tmp);                Tmp := SubBytes (Tmp);
Tmp := MixColumns (Tmp);
xmm1 := Tmp xor Round Key             xmm1 := Tmp xor Round Key
```

**Figure 9. The AESDEC and AESDECLAST Instructions**

```
AESDEC xmm1, xmm2/m128              AESDECLAST xmm1, xmm2/m128
Tmp := xmm1;                        State  := xmm1;
Round Key := xmm2/m128;             Round Key  := xmm2/m128
Tmp := InvShift Rows (Tmp);         Tmp := InvShift Rows (State);
Tmp := InvSubBytes (Tmp);           Tmp := InvSubBytes (Tmp);
Tmp := InvMix Columns (Tmp);
xmm1 := Tmp xor Round Key           xmm1 := :=  Tmp xor Round Key
```

## AESENC Example

**Figure 10. AESENC Example**

```
;; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key)
;; result delivered in xmm1
xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
AESENC result: a8311c2f9fdba3c58b104b58ded7e595
```

## AESENCLAST Example

**Figure 11. AESENCLAST Example**

```
;; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key)
;; result delivered in xmm1
xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
AESENCLAST result: c7fb881e938c5964177ec42553fdc611
```

## AESEDEC Example

**Figure 12. AESDEC Example**

```
;; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key)
;; result delivered in xmm1
xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
AESDEC result (in xmm1): 138ac342faea2787b58eb95eb730392a
```

## AESDECLAST Example

**Figure 13. AESDECLAST Example**

```
;; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key)
;; result delivered in xmm1
xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
AESDECLAST result (in xmm1): c5a391ef6b317f95d410637b72a593d0
```

## AES Flows in Terms of the AES Rounds Instructions

The AES encryption and decryption flows (see Figure 14 and Figure 15) can be expressed in terms of the four AES rounds instructions. As before, the color code is used for describing the respective different flows for the different key lengths (128/192/256

bits). In the following pseudo codes, it is assumed that the **10**, **12,** or **14** round keys are already derived (expanded) from the cipher key and stored, in the proper order, in an array (Round_Key_Encrypt and Round_Key_Decrypt). Further, it is also assumed that Round_Key_Encrypt [0] stores the first 128 bits of the cipher key (for "round #0" which is a simple xor operation).

**Figure 14. AES Encryption in Terms of AESENC/AESENCLAST**

```
Tmp = AddRoundKey (Data, Round_Key_Encrypt [0])
For round =  1-9 or 1-11 or 1-13:
  Tmp = AESENC (Tmp, Round_Key_Encrypt [round])
end loop
Tmp = AESENCLAST (Tmp, Round_Key_Encrypt [10 or 12 or 14])
Result = Tmp
```

**Figure 15. AES Decryption (Equivalent Inverse Cipher) in Terms of AESDEC/AESDECLAST**

```
Tmp = AddRoundKey (Data, Round_Key_Decrypt [0])
For round =  1-9 or 1-11 or 1-13:
  Tmp = AESDEC (Tmp, Round_Key_Decrypt [round])
end loop
Tmp = AESDECLAST (Tmp, Round_Key_Decrypt [10 or 12 or 14])
Result = Tmp
```

## AES-128 Encryption Example

The following code snippet shows an AES-128 encryption sequence.

**Figure 16. AES-128 Encryption Example**

```
        ;; Encryption sequence
        ;; data is in xmm1. Registers xmm2 – xmm1-xmm12 hold the round keys.
        ;; In the end - xmm1 holds the encryption result

        pxor xmm1, xmm2                    ; Round 0 (Round 0)
        aesenc xmm1, xmm3                  ; Round 1
        aesenc xmm1, xmm4                  ; Round 2
        aesenc xmm1, xmm5                  ; Round 3
        aesenc xmm1, xmm6                  ; Round 4
        aesenc xmm1, xmm7                  ; Round 5
        aesenc xmm1, xmm8                  ; Round 6
        aesenc xmm1, xmm9                  ; Round 7
        aesenc xmm1, xmm10                 ; Round 8
        aesenc xmm1, xmm11                 ; Round 9
        aesenclast xmm1, xmm12             ; Round 10
```

## AES-192 Decryption Example

The following code snippet shows an AES-192 encryption sequence.

**Figure 17. Using AESIMC for AES-192**

```
  ;; Decryption sequence
  ;; data is in xmm1. Registers xmm14 – xmm2 hold the round keys.
;; In the end - xmm1 holds the decryption result

        pxor xmm1, xmm14                   ; Round 0 (Round 0)
        aesdec xmm1, xmm13                 ; Round 1 (consume keys in reverse
order)
```

```
        aesdec xmm1, xmm12                  ; Round 2
        aesdec xmm1, xmm11                  ; Round 3
        aesdec xmm1, xmm10                  ; Round 4
        aesdec xmm1, xmm9                   ; Round 5
        aesdec xmm1, xmm8                   ; Round 6
        aesdec xmm1, xmm7                   ; Round 7
        aesdec xmm1, xmm6                   ; Round 8
        aesdec xmm1, xmm5                   ; Round 9
        aesdec xmm1, xmm4                   ; Round 10
        aesdec xmm1, xmm3                   ; Round 11
        aesdeclast xmm1, xmm2               ; Round 12
```

# AES Key Expansion

AES key generation is supported by two instructions. AESKEYGENASSIST is used for generating the round keys, used for encryption. AESIMC is used for converting the encryption round keys to a form usable for decryption.

## The AESKEYGENASSIST Instruction

**Figure 18. The AESKEYGENASSIST Instruction**

```
AESKEYGENASSIST xmm1, xmm2/m128, imm8
Tmp := xmm2/LOAD(m128)
X3[31-0] := Tmp[127-96];
X2[31-0] := Tmp[95-64];
X1[31-0] := Tmp[63-32];
X0[31-0] := Tmp[31-0];
RCON[7-0] := imm8;
xmm1 := [Rot (SubWord (X3)) ⊕ RCON, SubWord (X3), Rot (SubWord (X1)) ⊕ RCON,
SubWord (X1)]
```

Remark: "Doubleword ⊕ RCON" that is used in Figure 18 is stands for [Bye3, Bye2, Bye1, Bye0] XOR [0, 0, 0, RCON] (Byte3, Byte2, Byte1, Byte0 are the bytes of the corresponding doubleword).

## AESKEYGENASSIST Example

**Figure 19. AESKEYGENASSIST Example**

```
;; xmm2 holds a 128-bit input; imm8 holds the RCON value
;; result delivered in xmm1
xmm2 = 48692853686179295b477565726f6e5dl imm8 = 1
AESKEYGENASSIST result (in xmm1): 01eb848beb848a013424b5e524b5e434
```

## Key Schedule Generation Using AESKEYGENASSIST

The following code snippet shows an AES-128 key schedule preparation and decryption sequence. For other key lengths, please see the code examples provided in the subsequent chapter.

**Figure 20. AES-128 Key Expansion**

```
;#=========================
;# ;# Data Space Initialization ;#
;#=========================
$data->data(<<'DATA');
key       do 03c4fcf098815f7aba6d2ae2816157e2bh   ; 128 bit key (FIPS doc)
keyex_addr: keyex      do 00000000000000000000000000000000h   ;

        ;;
        ;; KEY SCHEDULE
        ;;
        movaps xmm1, OWORD PTR key    ; loading the key
        movaps OWORD PTR keyex, xmm1  ; Storing key in memory where all key
expansion
        mov rcx, OFFSET keyex_addr+16 ; setting store address for key expansion

        aeskeygenassist xmm2, xmm1, 0x1     ; Generating round key 1
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x2     ; Generating round key 2
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x4     ; Generating round key 3
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x8     ; Generating round key 4
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x10    ; Generating round key 5
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x20    ; Generating round key 6
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x40    ; Generating round key 7
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x80    ; Generating round key 8
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x1b    ; Generating round key 9
        call key_expansion_128
        aeskeygenassist xmm2, xmm1, 0x36    ; Generating round key 10
        call key_expansion_128

key_expansion_128:
        mov rdx, rcx
        pshufd xmm2, xmm2, 0b11111111
        pxor xmm2, xmm1
        movd eax, xmm2
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100101b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100110b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4
```

```
        pshufd xmm1, xmm1, 011100111b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movaps xmm1, OWORD PTR [rdx]
        ret
```

## Preparing the Decryption Round Keys Using AESIMC

AESIMC is used for preparing the round keys for decryption, using the Equivalent Inverse Cipher. The instruction is defined by the following pseudo code.

**Figure 21. The AESIMC Instruction**

```
   AESIMC xmm1, xmm2/m128
RoundKey := xmm2/m128;
xmm1 := InvMixColumns (RoundKey)
```

AESDEC and AESDECLAST carry out data decryption using the Equivalent Inverse Cipher. To feed the correct round keys, the decryption key schedule needs to be appropriately prepared. Assuming that the 10/12/14 round keys for encryption are already expanded and stored in an array, the decryption round keys can be easily prepared using the AESIMC instruction. More specifically, each one of the 10/12/14 encryption round keys should be filtered through AESIMC and the respective result becomes the decryption round key. The decryption algorithm consumes the round key in reverse order, and therefore the decryption round keys array needs to be stored in reverse order (compared with the encryption keys).

For example, consider AES-128, and assume that xmm2 holds round key #3. Then, after dispatching the instruction AESIMC xmm1, xmm2, the value of xmm1 would hold the decryption round key #8.

## AESIMC Example

**Figure 22. AESIMC Example**

```
;; xmm2 hold one 128-bit inputs (xmm2 = Round key)
;; result delivered in xmm1
xmm2 = 48692853686179295b477565726f6e5d
AESIMC result (in xmm1): 627a6f6644b109c82b18330a81c3b3e5
```

## Generating AES-192 Decryption Round Keys

The following code snippet shows an AES-192 key schedule preparation and decryption sequence.

**Figure 23. Using AESIMC for AES-192**

```
 ;;
kjd
;        The encryption round keys are in xmm2 through xmm14
;        The round keys are processed through aesimc
;        Decryption consumes the (transformed) round keys in reverse order
```

```
        ;;
        ;; DECRYPTION
        ;;

        aesimc xmm3, xmm3                   ; Generating keys for decryption
        aesimc xmm4, xmm4                   ; (passing through Inverse Mix Columns)
        aesimc xmm5, xmm5                   ;
        aesimc xmm6, xmm6                   ;
        aesimc xmm7, xmm7                   ;
        aesimc xmm8, xmm8                   ;
        aesimc xmm9, xmm9                   ;
        aesimc xmm10, xmm10                 ;
        aesimc xmm11, xmm11                 ;
        aesimc xmm12, xmm12                 ;
        aesimc xmm13, xmm13                 ;

        pxor xmm1, xmm14                    ; Round 0 (XORing)
        aesdec xmm1, xmm13                  ; Round 1 (consume keys in reverse
order)
        aesdec xmm1, xmm12                  ; Round 2
        aesdec xmm1, xmm11                  ; Round 3
        aesdec xmm1, xmm10                  ; Round 4
        aesdec xmm1, xmm9                   ; Round 5
        aesdec xmm1, xmm8                   ; Round 6
        aesdec xmm1, xmm7                   ; Round 7
        aesdec xmm1, xmm6                   ; Round 8
        aesdec xmm1, xmm5                   ; Round 9
        aesdec xmm1, xmm4                   ; Round 10
        aesdec xmm1, xmm3                   ; Round 11
        aesdeclast xmm1, xmm2               ; Round 12
```

## Application Programming Model

The AES extensions follow the same programming model as Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4 (see I*A-32 Intel Architecture Software Developer's Manual*, Volume 1). Operating systems that support handling Intel SSE state will also support applications that use the AES instructions. This is the same requirement for Legacy Intel SSE (i.e., Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4).

### Detecting AES Instructions

Before an application attempts to use the AES instructions (AESDEC, AESDECLAST, AESENC, AESENCLAST, AESIMC, AESKEYGENASSIST), it should verify that the processor supports these extensions. AES extensions is supported if CPUID.01H:ECX.AES[bit 25] = 1.

# Code Examples

This chapter provides three code examples for AES-128, AES-192, and AES-256. Each example shows the complete flow for the key expansion, encryption and decryption.

**Figure 24. AES-128 Key Expansion, Encryption and Decryption**

```
;#=========================
;# ;# Data Space Initialization ;#
;#=========================
$data->data(<<'DATA');
```

```
dec_data  do 0340737e0a29831318d305a88a8f64332h   ; data to encrypt (FIPS doc)
key       do 03c4fcf098815f7aba6d2ae2816157e2bh   ; 128 bit key (FIPS doc)
enc_data  do 0320b6a19978511dcfb09dc021d842539h   ; expected encryption result (FIPS
doc)
retdata   do 00000000000000000000000000000000h   ; where to store encrypted data

keyex_addr: keyex      do 00000000000000000000000000000000h   ;

DATA

;#==========================
;# ;# Main Code Segment ;#
;#==========================

$code->code(<<'CODE');

;; Enable use of MMX2 from software.
  mov rax, CR4               ; Set bit 9 (OSFXSR) and NOT bit 10
        or rax, 0200h              ; (OSMMEXCPT).
        mov CR4, rax

        mov eax, 01h                ; Verify support in CPU
        cpuid
        mov eax, 02000000h          ; Check if bit #25 is set
        and eax, ecx
;       jz FAIL

        ;;
        ;; KEY SCHEDULE
        ;;
        movaps xmm1, OWORD PTR key    ; loading the key
        movaps OWORD PTR keyex, xmm1  ; Storing key in memory where all key
expansion
        mov rcx, OFFSET keyex_addr+16 ; setting store address for key expansion

        aeskeygen xmm2, xmm1, 0x1     ; Generating round key 1
        call key_expansion
        aeskeygen xmm2, xmm1, 0x2     ; Generating round key 2
        call key_expansion
        aeskeygen xmm2, xmm1, 0x4     ; Generating round key 3
        call key_expansion
        aeskeygen xmm2, xmm1, 0x8     ; Generating round key 4
        call key_expansion
        aeskeygen xmm2, xmm1, 0x10    ; Generating round key 5
        call key_expansion
        aeskeygen xmm2, xmm1, 0x20    ; Generating round key 6
        call key_expansion
        aeskeygen xmm2, xmm1, 0x40    ; Generating round key 7
        call key_expansion
        aeskeygen xmm2, xmm1, 0x80    ; Generating round key 8
        call key_expansion
        aeskeygen xmm2, xmm1, 0x1b    ; Generating round key 9
        call key_expansion
        aeskeygen xmm2, xmm1, 0x36    ; Generating round key 10
        call key_expansion

        ;;
        ;; PERFORMING ENCRYPTION
        ;;

        movaps xmm1, OWORD PTR dec_data       ; loading the data to be encrypted
        movapd xmm2, OWORD PTR keyex          ; loading the keys for the encryption
        movapd xmm3, OWORD PTR [keyex+0x10]   ;
```

```
        movapd xmm4, OWORD PTR [keyex+0x20]   ;
        movapd xmm5, OWORD PTR [keyex+0x30]   ;
        movapd xmm6, OWORD PTR [keyex+0x40]   ;
        movapd xmm7, OWORD PTR [keyex+0x50]   ;
        movapd xmm8, OWORD PTR [keyex+0x60]   ;
        movapd xmm9, OWORD PTR [keyex+0x70]   ;
        movapd xmm10, OWORD PTR [keyex+0x80]  ;
        movapd xmm11, OWORD PTR [keyex+0x90]  ;
        movapd xmm12, OWORD PTR [keyex+0xA0]  ;

        pxor xmm1, xmm2                     ; Round 0 (XORing)
        aesenc xmm1, xmm3                   ; Round 1
        aesenc xmm1, xmm4                   ; Round 2
        aesenc xmm1, xmm5                   ; Round 3
        aesenc xmm1, xmm6                   ; Round 4
        aesenc xmm1, xmm7                   ; Round 5
        aesenc xmm1, xmm8                   ; Round 6
        aesenc xmm1, xmm9                   ; Round 7
        aesenc xmm1, xmm10                  ; Round 8
        aesenc xmm1, xmm11                  ; Round 9
        aesenclast xmm1, xmm12              ; Round 10

        ;;
        ;; VERIFYING CORRECTNESS
        ;;

        movups OWORD PTR retdata, xmm1      ; Storing the encrypted data
        mov rax, QWORD PTR retdata          ;
        cmp rax, QWORD PTR enc_data         ; Comparing the MSB of the encrypted
data
        jne FAIL
        mov rax, QWORD PTR [retdata+8]      ;
        cmp rax, QWORD PTR [enc_data+8]     ; Comparing the LSB of the encrypted
data
        jne FAIL

;; PERFORMING DECRYPTION
        aesimc xmm3, xmm3                   ; Generating keys for decryption
        aesimc xmm4, xmm4                   ;
        aesimc xmm5, xmm5                   ;
        aesimc xmm6, xmm6                   ;
        aesimc xmm7, xmm7                   ;
         aesimc xmm8, xmm8                   ;
        aesimc xmm9, xmm9                   ;
        aesimc xmm10, xmm10                 ;
        aesimc xmm11, xmm11                 ;

        pxor xmm1, xmm12                    ; XORing
        aesdec xmm1, xmm11                  ; Round 1
        aesdec xmm1, xmm10                  ; Round 2
        aesdec xmm1, xmm9                   ; Round 3
        aesdec xmm1, xmm8                   ; Round 4
        aesdec xmm1, xmm7                   ; Round 5
        aesdec xmm1, xmm6                   ; Round 6
        aesdec xmm1, xmm5                   ; Round 7
        aesdec xmm1, xmm4                   ; Round 8
        aesdec xmm1, xmm3                   ; Round 9
        aesdeclast xmm1, xmm2               ; Round 10

;;
;; VERIFYING CORRECTNESS
;;
```

```
movdqu OWORD PTR retdata, xmm1      ; Storing the new decrypted data
mov rax, QWORD PTR retdata          ;
cmp rax, QWORD PTR dec_data         ; Comparing MSB of the decrypted data with
original data
jne FAIL
mov rax, QWORD PTR [retdata+8]      ;
cmp rax, QWORD PTR [dec_data+8]     ; Comparing LSB of the decrypted data with
original data
jne FAIL


        mov ebx, 0acedh
        hlt;

key_expansion:

        mov rdx, rcx
        pshufd xmm2, xmm2, 0b11111111
        pxor xmm2, xmm1
        movd eax, xmm2
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100101b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100110b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100111b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movaps xmm1, OWORD PTR [rdx]
        ret

FAIL:
        mov ebx, 0deadh
        hlt
CODE
```

**Figure 25. AES-192 Key Expansion, Encryption and Decryption**

```
;#========================== ;# ;# Data Space Initialization ;#
;#==========================

$data->data(<<'DATA');
dec_data  do 0ffeeddccbbaa99887766554433221100h   ; data to encrypt (FIPS doc)
key       do 00f0e0d0c0b0a090807060504030201 00h   ; 192 bit key (FIPS doc)
key_      do 0000000000000000001716151413121110h
enc_data  do 091710deca070af6ee0df4c86a47ca9ddh   ; expected encryption result (FIPS
doc)
retdata   do 00000000000000000000000000000000h   ; where to stored encrypted data

keyex_addr: keyex      do 00000000000000000000000000000000h   ;
```

```
DATA

;#========================== ;# ;# Main Code Segment ;#
;#==========================

;; Enable use of MMX2 from software.
        mov rax, CR4               ; Set bit 9 (OSFXSR) and NOT bit 10
        or rax, 0200h              ; (OSMMEXCPT).
        mov CR4, rax
        mov eax, 01h                ; Verify support in CPU
        cpuid
        mov eax, 02000000h          ; Check if bit #25 is set
        and eax, ecx
;       jz FAIL

        ;;
        ;; KEY SCHEDULE
        ;;

        movdqu xmm1, OWORD PTR key    ; loading the key
        movq xmm3, QWORD PTR key_
        movdqu OWORD PTR keyex, xmm1  ; Storing key in memory with all key expansion
        movq QWORD PTR [keyex + 0x10], xmm3
        mov rcx, OFFSET keyex_addr+24 ; setting store address for key expansion

        aeskeygen xmm2, xmm3, 0x1     ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x2     ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x4     ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x8     ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x10    ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x20    ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x40    ;  expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x80    ;  expanding the key
        call key_expansion

        ;;
        ;; PERFORMING ENCRYPTION
        ;;

        movdqu xmm1, OWORD PTR dec_data       ; loading the data to be encrypted
        movdqu xmm2, OWORD PTR keyex          ; loading the keys for the encryption
        movdqu xmm3, OWORD PTR [keyex+0x10]   ;
        movdqu xmm4, OWORD PTR [keyex+0x20]   ;
        movdqu xmm5, OWORD PTR [keyex+0x30]   ;
        movdqu xmm6, OWORD PTR [keyex+0x40]   ;
        movdqu xmm7, OWORD PTR [keyex+0x50]   ;
        movdqu xmm8, OWORD PTR [keyex+0x60]   ;
        movdqu xmm9, OWORD PTR [keyex+0x70]   ;
        movdqu xmm10, OWORD PTR [keyex+0x80]  ;
        movdqu xmm11, OWORD PTR [keyex+0x90]  ;
        movdqu xmm12, OWORD PTR [keyex+0xA0]  ;
        movdqu xmm13, OWORD PTR [keyex+0xB0]  ;
        movdqu xmm14, OWORD PTR [keyex+0xC0]  ;

        pxor xmm1, xmm2                    ; XORing
```

```
        aesenc xmm1, xmm3                   ; Round 1
        aesenc xmm1, xmm4                   ; Round 2
        aesenc xmm1, xmm5                   ; Round 3
        aesenc xmm1, xmm6                   ; Round 4
        aesenc xmm1, xmm7                   ; Round 5
        aesenc xmm1, xmm8                   ; Round 6
        aesenc xmm1, xmm9                   ; Round 7
        aesenc xmm1, xmm10                  ; Round 8
        aesenc xmm1, xmm11                  ; Round 9
        aesenc xmm1, xmm12                  ; Round 10
        aesenc xmm1, xmm13                  ; Round 11
        aesenclast xmm1, xmm14              ; Round 12

        ;;
        ;; VERIFYING CORRECTNESS
        ;;

        movdqu OWORD PTR retdata, xmm1      ; Storing the new decrypted data
        mov rax, QWORD PTR retdata          ;
        cmp rax, QWORD PTR enc_data         ; Comparing the MSB of the encrypted
data
                                            ; to the expected result
        jne FAIL
        mov rax, QWORD PTR [retdata+8]      ;
        cmp rax, QWORD PTR [enc_data+8]     ; Comparing the LSB of the encrypted
data
                                            ; to the expected result
        jne FAIL

        ;;
        ;; PERFORMING DECRYPTION
        ;;

        aesimc xmm3, xmm3                   ; Generating keys for decryption
        aesimc xmm4, xmm4                   ; (passing through Inverse Mix Columns)
        aesimc xmm5, xmm5                   ;
        aesimc xmm6, xmm6                   ;
        aesimc xmm7, xmm7                   ;
        aesimc xmm8, xmm8                   ;
        aesimc xmm9, xmm9                   ;
        aesimc xmm10, xmm10                 ;
        aesimc xmm11, xmm11                 ;
        aesimc xmm12, xmm12                 ;
        aesimc xmm13, xmm13                 ;

        pxor xmm1, xmm14                    ; Round 0 (XORing)
        aesdec xmm1, xmm13                  ; Round 1 (consume keys in reverse
order)
        aesdec xmm1, xmm12                  ; Round 2
        aesdec xmm1, xmm11                  ; Round 3
        aesdec xmm1, xmm10                  ; Round 4
        aesdec xmm1, xmm9                   ; Round 5
        aesdec xmm1, xmm8                   ; Round 6
        aesdec xmm1, xmm7                   ; Round 7
        aesdec xmm1, xmm6                   ; Round 8
        aesdec xmm1, xmm5                   ; Round 9
        aesdec xmm1, xmm4                   ; Round 10
        aesdec xmm1, xmm3                   ; Round 11
        aesdeclast xmm1, xmm2               ; Round 12

        ;;
        ;; VERIFYING CORRECTNESS
        ;;

        movdqu OWORD PTR retdata, xmm1      ; Storing the new decrypted data
```

```
        mov rax, QWORD PTR retdata          ;
        cmp rax, QWORD PTR dec_data         ; Comparing the MSB of the (new)
decrypted data
                                                    ; to the original data
        jne FAIL
        mov rax, QWORD PTR [retdata+8]      ;
        cmp rax, QWORD PTR [dec_data+8]     ; Comparing the LSB of the (new)
decrypted data
                                                   ; to the original data
        jne FAIL

        mov ebx, 0acedh
        hlt;

key_expansion:

        mov rdx, rcx
        pshufd xmm2, xmm2, 001010101b
        pxor xmm2, xmm1
        movd eax, xmm2
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100101b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100110b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100111b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movd ebx, xmm3
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm3, xmm3, 011100101b
        movd ebx, xmm3
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movdqu xmm1, OWORD PTR [rdx]
        add rdx, 0x10
        movq xmm3, QWORD PTR [rdx]
        ret

FAIL:
        mov ebx, 0deadh
        hlt
CODE
```

**Figure 26. AES-256 Key Expansion, Encryption and Decryption**

```
;#========================== ;# ;# Data Space Initialization ;#
;#==========================

$data->data(<<'DATA');
dec_data  do 0ffeeddccbbaa99887766554433221100h   ; data to encrypt (FIPS doc)
key       do 00f0e0d0c0b0a0908070605040302010 0h   ; 256 bit key (FIPS doc)
key_      do 01f1e1d1c1b1a19181716151413121110h
enc_data  do 08960494b9049fceabf456751cab7a28eh   ; expected encryption result (FIPS
doc)

retdata   do 00000000000000000000000000000000h   ; where to store the encrypted
data
keyex_addr: keyex     do 00000000000000000000000000000000h   ;

DATA

;#========================== ;# ;# Main Code Segment ;#
;#==========================

$code->code(<<'CODE');

;; Enable use of MMX2 from software.
        mov rax, CR4               ; Set bit 9 (OSFXSR) and NOT bit 10
        or rax, 0200h              ; (OSMMEXCPT).
        mov CR4, rax

        mov eax, 01h                ; Verify support in CPU
        cpuid
        mov eax, 02000000h          ; Check if bit #25 is set
        and eax, ecx
;       jz FAIL

        ;;
        ;; KEY SCHEDULE
        ;;

        movdqu xmm1, OWORD PTR key    ; loading the key
        movdqu xmm3, OWORD PTR key_
        movdqu OWORD PTR keyex, xmm1  ; Storing key in memory with all key expansion
        movdqu OWORD PTR [keyex + 0x10], xmm3
        mov rcx, OFFSET keyex_addr+0x20 ; setting store address for key expansion

        aeskeygen xmm2, xmm3, 0x1     ; Expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x2     ; Expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x4     ; Expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x8     ; Expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x10    ; Expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x20    ; Expanding the key
        call key_expansion
        aeskeygen xmm2, xmm3, 0x40    ; Expanding the key
        call key_expansion

        ;;
        ;; PERFORMING ENCRYPTION
        ;;

        movdqu xmm1, OWORD PTR dec_data       ; loading the data to be encrypted
```

```
        movdqu xmm2, OWORD PTR [keyex+0x10]   ;
        movdqu xmm3, OWORD PTR [keyex+0x20]   ;
        movdqu xmm4, OWORD PTR [keyex+0x30]   ;
        movdqu xmm5, OWORD PTR [keyex+0x40]   ;
        movdqu xmm6, OWORD PTR [keyex+0x50]   ;
        movdqu xmm7, OWORD PTR [keyex+0x60]   ;
        movdqu xmm8, OWORD PTR [keyex+0x70]   ;
        movdqu xmm9, OWORD PTR [keyex+0x80]   ;
        movdqu xmm10, OWORD PTR [keyex+0x90]  ;
        movdqu xmm11, OWORD PTR [keyex+0xA0]  ;
        movdqu xmm12, OWORD PTR [keyex+0xB0]  ;
        movdqu xmm13, OWORD PTR [keyex+0xC0]  ;
        movdqu xmm14, OWORD PTR [keyex+0xD0]  ;
        movdqu xmm15, OWORD PTR [keyex+0xE0]  ;

        pxor xmm1, OWORD PTR keyex          ; Round 0 (XORing)
        aesenc xmm1, xmm2                   ; Round 1
        aesenc xmm1, xmm3                   ; Round 2
        aesenc xmm1, xmm4                   ; Round 3
        aesenc xmm1, xmm5                   ; Round 4
        aesenc xmm1, xmm6                   ; Round 5
        aesenc xmm1, xmm7                   ; Round 6
        aesenc xmm1, xmm8                   ; Round 7
        aesenc xmm1, xmm9                   ; Round 8
        aesenc xmm1, xmm10                  ; Round 9
        aesenc xmm1, xmm11                  ; Round 10
        aesenc xmm1, xmm12                  ; Round 11
        aesenc xmm1, xmm13                  ; Round 12
        aesenc xmm1, xmm14                  ; Round 13
        aesenclast xmm1, xmm15              ; Round 14

        ;;
        ;; VERIFYING CORRECTNESS
        ;;

        movdqu OWORD PTR retdata, xmm1      ; Storing the new encrypted data
        mov rax, QWORD PTR retdata          ;
        cmp rax, QWORD PTR enc_data         ; Comparing the MSB of the encrypted
data
                                                    ; to the expected result
        jne FAIL
        mov rax, QWORD PTR [retdata+8]      ;
        cmp rax, QWORD PTR [enc_data+8]     ; Comparing the LSB of the encrypted
data
                                                    ; to the expected result
        jne FAIL

        ;;
        ;; PERFORMING DECRYPTION
        ;;

        aesimc xmm2, xmm2                   ; Generating keys for decryption
        aesimc xmm3, xmm3                   ; (passing through Inverse Mix Columns)
        aesimc xmm4, xmm4                   ;
        aesimc xmm5, xmm5                   ;
        aesimc xmm6, xmm6                   ;
        aesimc xmm7, xmm7                   ;
        aesimc xmm8, xmm8                   ;
        aesimc xmm9, xmm9                   ;
        aesimc xmm10, xmm10                 ;
        aesimc xmm11, xmm11                 ;
        aesimc xmm12, xmm12                 ;
        aesimc xmm13, xmm13                 ;
        aesimc xmm14, xmm14                 ;
```

```
        pxor xmm1, xmm15                    ; XORing
        aesdec xmm1, xmm14                  ; Round 1  (consume keys in reverse
order)
        aesdec xmm1, xmm13                  ; Round 2
        aesdec xmm1, xmm12                  ; Round 3
        aesdec xmm1, xmm11                  ; Round 4
        aesdec xmm1, xmm10                  ; Round 5
        aesdec xmm1, xmm9                   ; Round 6
        aesdec xmm1, xmm8                   ; Round 7
        aesdec xmm1, xmm7                   ; Round 8
        aesdec xmm1, xmm6                   ; Round 9
        aesdec xmm1, xmm5                   ; Round 10
        aesdec xmm1, xmm4                   ; Round 11
        aesdec xmm1, xmm3                   ; Round 12
        aesdec xmm1, xmm2                   ; Round 13
        aesdeclast xmm1, OWORD PTR keyex    ; Round 14

        ;;
        ;; VERIFYING CORRECTNESS
        ;;

        movdqu OWORD PTR retdata, xmm1      ; Storing the new decrypted data
        mov rax, QWORD PTR retdata          ;
        cmp rax, QWORD PTR dec_data         ; Comparing the MSB of the (new)
decrypted data
                                                    ; with the original data
        jne FAIL
        mov rax, QWORD PTR [retdata+8]      ;
        cmp rax, QWORD PTR [dec_data+8]     ; Comparing the LSB of the new decrypted
data
                                                    ; with the original data
        jne FAIL

        mov ebx, 0acedh
        hlt;

key_expansion:

        mov rdx, rcx
        pshufd xmm2, xmm2, 011111111b
        pxor xmm2, xmm1
        movd eax, xmm2
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100101b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100110b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100111b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movdqu xmm4, OWORD PTR [rdx]
```

```
        aeskeygen xmm4, xmm4, 0
        pshufd xmm4, xmm4, 011100110b
        movd eax, xmm4
        movd ebx, xmm3
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm3, xmm3, 011100101b
        movd ebx, xmm3
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm3, xmm3, 011100110b
        movd ebx, xmm3
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm3, xmm3, 011100111b
        movd ebx, xmm3
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movdqu xmm1, OWORD PTR [rdx]
        add rdx, 0x10
        movdqu xmm3, OWORD PTR [rdx]
        ret

FAIL:
        mov ebx, 0deadh
        hlt
CODE
```

# Using AES-NI with Parallel Modes of Operation

This chapter explains how throughput can be enhanced with AES using parallel modes of operation. Consider the code snippet described in Figure 7, for encrypting AES-128 in ECB mode. In this example, there are 8 data blocks in xmm2-xmm9, and a Round Key is loaded into xmm1. For each round, 8 AES round instructions are dispatched, operating on the 8 data blocks with the same Round Key. Then, the next round key is loaded. The 8 blocks encryption results are eventually stored into memory, ready load a new set of 8 data blocks. This way, the program encrypts 8 data blocks simultaneously, but the order is different from the order shown in the previous chapters. Instead of completing the encryption of one block and then continuing to the next block, the code computes one AES round on all 8 blocks, using one Round Key, and then continues to the next round (using the next Round Key). This "loop-reversal" technique is applicable to any parallel mode of operation such as CTR and CBC decrypt (but not for CBC encrypt).

The underlying fully-pipelined hardware implies that AES instructions could be dispatched "each cycle" if data is available. In a parallel mode of operation, using the AES instructions and loop-reversing software, data can indeed be made available in (almost) every cycle.

The following rough performance estimate illustrates the performing gain (neglecting loads and stores). Suppose that the latency of the AES instructions is L cycles, and L ≤ 8 (the actual latency of the AES instructions is L=6 cycles and this example uses 8 xmm registers). Then, encryption of the 8 data blocks would be completed after (roughly) 88+L cycles (pxor is done within 1 cycle). Therefore, the obtained throughput is around (88+6)/8=12 cycles per block (16B), which approaches the theoretical throughput limit. This simplified estimate ignores several factors (e.g., loads/stores), but nevertheless, the measured effect is quite close to the estimated one.

**Figure 27. AES-128 Parallel Modes of Operation**

```
;#========================== ;# ;# Data Space Initialization ;#
;#==========================

$data->data(<<'DATA'); datablock                    ; This is data to be encrypted
          do 0340737e0a29831318d305a88a8f64331h   ;
          do 0340737e0a29831318d305a88a8f64332h   ;
          do 0340737e0a29831318d305a88a8f64333h   ;
          do 0340737e0a29831318d305a88a8f64334h   ;
          do 0340737e0a29831318d305a88a8f64335h   ;
          do 0340737e0a29831318d305a88a8f64336h   ;
          do 0340737e0a29831318d305a88a8f64337h   ;
          do 0340737e0a29831318d305a88a8f64338h   ;

key       do 03c4fcf098815f7aba6d2ae2816157e2bh   ; 128 bit key

retdata   do 00000000000000000000000000000000h   ; where to store encrypted data

keyex_addr: keyex      do 00000000000000000000000000000000h   ;

DATA

;#========================== ;# ;# Main Code Segment ;#
;#==========================

$code->code(<<'CODE');

;; Enable use of MMX2 from software.
        mov rax, CR4                ; Set bit 9 (OSFXSR) and NOT bit 10
        or rax, 0200h               ; (OSMMEXCPT).
        mov CR4, rax

        mov eax, 01h                 ; Verify support in CPU
        cpuid
        mov eax, 02000000h           ; Check if bit #25 is set
        and eax, ecx
;       jz FAIL

        ;; GENERATING KEY SCHEDULE

        movdqu xmm1, OWORD PTR key    ; loading the key
        movdqu OWORD PTR keyex, xmm1  ; Store key in memory where all round keys are
stored
        mov rcx, OFFSET keyex_addr+16 ; setting store address for key expansion

        aeskeygen xmm2, xmm1, 0x1     ; Generating round key 1
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x2     ; Generating round key 2
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x4     ; Generating round key 3
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x8     ; Generating round key 4
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x10    ; Generating round key 5
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x20    ; Generating round key 6
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x40    ; Generating round key 7
        call key_expansion_128
```

```
        aeskeygen xmm2, xmm1, 0x80    ; Generating round key 8
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x1b    ; Generating round key 9
        call key_expansion_128
        aeskeygen xmm2, xmm1, 0x36    ; Generating round key 10
        call key_expansion_128

        ;;
        ;; PERFORMING ENCRYPTION

        mov rdx, OFFSET keyex_addr
        movdqu xmm1, OWORD PTR [rdx]           ; loading the key for the encryption
        movdqu xmm2, OWORD PTR [datablock]          ; loading entire data block
        movdqu xmm3, OWORD PTR [datablock+0x10]     ;
        movdqu xmm4, OWORD PTR [datablock+0x20]     ;
        movdqu xmm5, OWORD PTR [datablock+0x30]     ;
        movdqu xmm6, OWORD PTR [datablock+0x40]     ;
        movdqu xmm7, OWORD PTR [datablock+0x50]     ;
        movdqu xmm8, OWORD PTR [datablock+0x60]     ;
        movdqu xmm9, OWORD PTR [datablock+0x70]     ;

        pxor xmm2, xmm1                     ; XORing
        pxor xmm3, xmm1                     ; XORing
        pxor xmm4, xmm1                     ; XORing
        pxor xmm5, xmm1                     ; XORing
        pxor xmm6, xmm1                     ; XORing
        pxor xmm7, xmm1                     ; XORing
        pxor xmm8, xmm1                     ; XORing
        pxor xmm9, xmm1                     ; XORing

        mov ecx, 9

main_loop:

        add rdx, 0x10
        movdqu xmm1, OWORD PTR [rdx]  ; loading the key for the encryption

        aesenc xmm2, xmm1                   ; Encrypting
        aesenc xmm3, xmm1                   ;
        aesenc xmm4, xmm1                   ;
        aesenc xmm5, xmm1                   ;
        aesenc xmm6, xmm1                   ;
        aesenc xmm7, xmm1                   ;
        aesenc xmm8, xmm1                   ;
        aesenc xmm9, xmm1                   ;

        loop main_loop

        add rdx, 0x10
        movdqu xmm1, OWORD PTR [rdx]  ; loading the key for the encryption

        aesenclast xmm2, xmm1              ; Last round
        aesenclast xmm3, xmm1              ;
        aesenclast xmm4, xmm1              ;
        aesenclast xmm5, xmm1              ;
        aesenclast xmm6, xmm1              ;
        aesenclast xmm7, xmm1              ;
        aesenclast xmm8, xmm1              ;
        aesenclast xmm9, xmm1              ;


        movdqu OWORD PTR [datablock], xmm2          ; storing the encrypted block
        movdqu OWORD PTR [datablock+0x10], xmm3     ;
        movdqu OWORD PTR [datablock+0x20], xmm4       ;
        movdqu OWORD PTR [datablock+0x30], xmm5       ;
        movdqu OWORD PTR [datablock+0x40], xmm6       ;
        movdqu OWORD PTR [datablock+0x50], xmm7       ;
        movdqu OWORD PTR [datablock+0x60], xmm8       ;
        movdqu OWORD PTR [datablock+0x70], xmm9       ;

        mov ebx, 0acedh
```

```
        hlt;

key_expansion_128:

        mov rdx, rcx
        pshufd xmm2, xmm2, 0b11111111
        pxor xmm2, xmm1
        movd eax, xmm2
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100101b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100110b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        pshufd xmm1, xmm1, 011100111b
        movd ebx, xmm1
        xor eax, ebx
        mov DWORD PTR [rcx], eax
        add rcx, 4

        movdqu xmm1, OWORD PTR [rdx]
        ret

FAIL:
        mov ebx, 0deadh
        hlt
CODE
```

# Summary

This paper described Intel's new AES instructions which will be part of the Intel architecture in all processors as of the 2009 generation. AES is the leading standard for symmetric encryption, used in a variety of applications, and the raising demands for platform security and privacy make its usage ubiquitous. Consequently, a high performance and secure solution for AES computations in commodity processors is an important technology. The presented AES architecture consists of six instructions to support encryption, decryption and Key Expansion, and the paper provided detailed information on their software usage. Together, this provides a comprehensive hardware solution for AES, offering high performance, software flexibility, and security against side channel attacks.

# About the Author

**Shay Gueron** is a Security Architect in the Mobility Group at Intel Corporation, working at the Israel Design Center. His interests include applied security, cryptography, and algorithms. He holds a Ph.D. in applied mathematics from Technion—Israel Institute of Technology, and is also an Associate Professor at the Department of Mathematics, Faculty of Science and Science Education, at the University of Haifa, Israel.